Technical Cybersecurity

Explaining the Call Stack

What is the stack?

Statically Allocated Storage

- In the program memory image
- Fast
- No allocation/deallocation required
- Created by the compiler (if C) or the programmer based on particular conventions
- Certain instructions will use the stack (**ret**, **push**, **pop**)

How does it work?

LIFO Queue

- push, pop semantics
- Controlled and referenced by the stack pointer and base pointer
    - x86 pushes return addresses on stack
    - x86_32 stored function args on stack
    - x86_64 doesn’t
    - MIPS, ARM? Kinda depends.

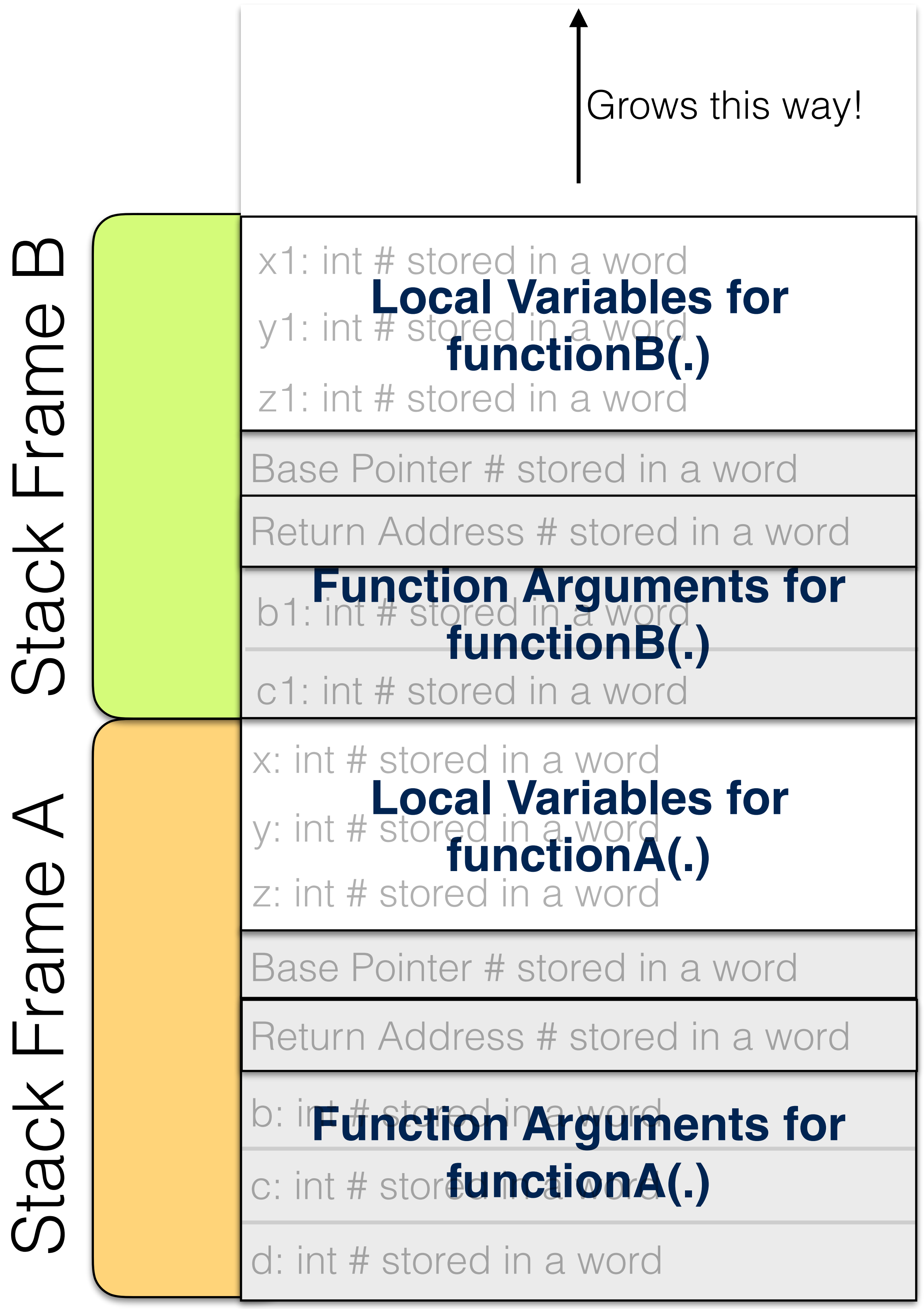
A Stack Example

LOTS OF DATA!

- Function arguments
- Return address
- Local variables

FREQUENTLY HIGH -> LOW

- Doesn't always


Grows this way!
Stack Frame B
Stack Frame A
x1: int # stored in a word
y1: int # stored in a word
z1: int # stored in a word
Local Variables for functionB(.)
Base Pointer # stored in a word
Return Address # stored in a word
b1: int # stored in a word
c1: int # stored in a word
Function Arguments for functionB(.)
x: int # stored in a word
y: int # stored in a word
z: int # stored in a word
Local Variables for functionA(.)
Base Pointer # stored in a word
Return Address # stored in a word
b: int # stored in a word
c: int # stored in a word
d: int # stored in a word
Function Arguments for functionA(.)

Calling Conventions

WHO PUTS WHAT WHERE

- Many of them, common ones include stdcall, cdecl, syscall, etc.

CDECL (C PROGRAMMING LANGUAGE, GCC)

- arguments passed on stack
- arguments pushed in right-to-left order
- caller cleans and sets up the stack frame

Next up: disassembly!